SICS/R-89/8909

STRIPS-Like Planning Using GCLA

by
Martin Aronsson

SICS research report
R89009
ISSN 0283-3638

SICS/R-89/8909

STRIPS-Like Planning Using GCLA

by

Martin Aronsson
Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, Sweden
email:martin@sics.se

Tel: +46 8 752 15 00

Abstract
This report describes how a STRIPS planning system can be implemented in GCLA. A short introduction to the problem of the blocks world is given, as well as an introduction to STRIPS. Then the GCLA program implementing the planning system is presented, together with queries concerning planning and simulation problems.

Contents:

1. Introduction
2. The Blocks World
3. STRIPS
4. The Language GCLA
5 The GCLA Program
6. Planning
7. Simulation
8. Conclusion

References
Appendix

1. Introduction

The work in this report started with a number of sessions with the Swedish company Ellemtel, trying to specify a telephone exchange. The specification was written in GCLA as a number of transitions, which looked very much as the STRIPS planning system. From that program the results in this paper have their roots.

The idea used in this GCLA program is very simple: an *action* is a function from one *state* to another. A *state* is represented by a number of clauses in the program, and as the execution proceeds, these clauses are rewritten, removed or added by the actions. Making the plan becomes a form of equation solving; find an action (sequence of actions) such that evaluating it takes us from the current state to the desired state. Simulation becomes a form of function evaluation; evaluate the function call and see what the state becomes. With this view, planning or simulation becomes a matter of instantiation pattern in the query.

Much of the work in this paper has been carried out in cooperation with Anette Gäredal, to whom I am very thankful. Much of the basic program has been carried out together with her.

2. The Blocks World

The blocks world is a simple example, which is very often used in planning problems. It is very easy to understand, yet involves many problems that must be dealt with in a proper way in order to derive sound solutions. The blocks world consists of a number of blocks, a table and a robot arm. The blocks can be stacked on each other, or put down on the table, which has infinitely many places to put a block down on. The problem is to find a number of actions which reorder the blocks in order to reach a state from another state. For example, the problem could be to find a list of actions to reorder the pile of blocks in fig. 1 to the state showed in fig. 2.

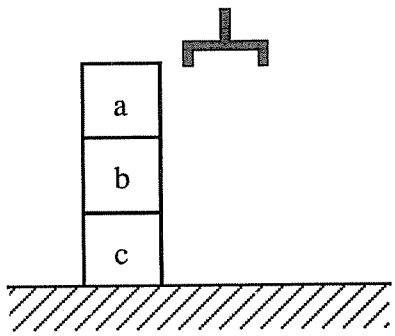

fig. 1

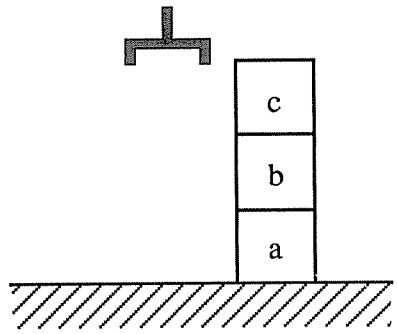

fig. 2

A state (configuration) in the blocks world could be described with the predicates *table(X)*, *on(X,Y)* and *clear(X)*. For example, the configuration in fig. 1 could be described by the four statements:

clear(a), on(a,b), on(b,c), table(c)

There are a lot of possible actions that a robot can perform with a pile of blocks. We will stick to three possible actions, namely *stack*, *unstack*, and *move*. The action *stack* picks a block from the table and puts it onto another block. The action *unstack* performs the opposite, that is, picks the topmost block from a pile of blocks and puts it down on the table. The action *move* moves the topmost block from one pile to another. This can be formulated as if-then rules (we assume that X and Y are distinct variables):

stack(X,Y):
if on table(X) and clear(X) and clear(Y),
then not on table(X) and not clear(Y) and on(X,Y).

unstack(X,Y):
if on(X,Y) and clear(X),
then on table(X) and clear(Y) and not on(X,Y).
move(X,Y,Z):
if clear(X) and clear(Z) and on(X,Y),
then on(X,Z) and clear(Y) and not clear(Z)

These robot actions thus change one state into another, by changing the state destructively. This should be contrasted with logical implication, where a global database is updated with the new information from a true implication, but nothing is ever removed from the database. A solution using situation calculus and frame axioms is given in [Genesereth & Nilsson].

3. STRIPS

STRIPS was invented by Fikes and Nilsson in 1971, and is described in a number of places in literature, for example see [Nilsson]. STRIPS's rules is divided into three parts: a precondition list, an adding list and a deleting list. The precondition list (*P*) contains the facts that must hold in order for this action to be performable. The adding list (*A*) contains facts that becomes true by applying the rule, and the deleting list (*D*) contains facts that become false by applying the rule. The general format for a STRIPS rule is *if P then delete D from the current state and add A to the current state*. The rules mentioned in section 2 are then formulated as:

stack(X,Y):
P: clear(X), clear(Y), table(X)
D: clear(Y),table(X)
A: on(X,Y)
unstack(X,Y):
P: clear(X), on(X,Y)
D: on(X,Y)
A: clear(Y), table(X)

move(X,Y,Z):

P: clear(X), clear(Z), on(X,Y)

D: on(X,Y), clear(Z)

A: on(X,Z), clear(Y)

For example, one of the possible sequences of STRIPS actions for going from the state in fig. 1 to the state in fig. 2 is {*unstack(a,b), move(b,c,a), stack(c,b)*}.

4. GCLA

GCLA [Aronsson & Hallnäs, Hallnäs & Schroeder-Heister, Aronsson] is a new programming language, especially suited for hypothetical reasoning. The language is best regarded as a logic programming language, with some properties usually found among functional languages.

In GCLA, a program is not looked upon as a set of true facts and implications from which conclusions are drawn using a specific computation rule (for example SLD resolution as in Prolog), but rather as a definition of rules determining the possible inferences. In a sense the program defines the "systems mind", meaning that the inference rules of the logic are given by the program. To execute a program, a query G is posed to the program, asking whether there is a substitution σ such that $G\sigma$ holds according to the logic defined by the program. The Goal G has the form $\Gamma \vdash c$, where Γ is a list of assumptions, and c is the conclusion from the assumption Γ. The system tries to construct a deduction showing that $G\sigma$ holds in the given logic.

The clauses in the program could be used both for forming the introduction rules, applicable to the consequence of the goal (to the right of $\vdash$), and for forming the elimination rules, applicable to the assumptions of a goal (to the left of $\vdash$) [Hallnäs & Schroeder-Heister]. The introduction rules of the logic (defined by the GCLA program) are the clauses as they stand, and the elimination rules are formed by the largest unifiable set of clauses in the program, which are unifiable with the considered assumption.

For example, if the program is {`p :- q, q :- r, q :- s`}, and we ask if `r` follows from `p` (`p` $\vdash$ `r`), the GCLA interpreter will answer `yes`, because the clause `q :- r` is used as an introduction rule, and the clause `p :- q` could be used either as an introduction rule or an elimination rule. In the first case we will have the goal `p` $\vdash$ `p`, in the second case we will have the goal `q` $\vdash$ `q`, both trivially provable.

There is an essential conceptual difference between assumptions and definitional clauses. Consider the program {`w :- q`}. The query `w` $\vdash$ `q` succeeds, because using the rule `w :- q` as an elimination rule gives us the goal `q` $\vdash$ `q`, which succeeds. But, if the program is changed to {`w :- true, w :- q`} (where `w :- true` stands for a rule which is unconditionally true), that is, asserting the fact `w`, the goal $\vdash$ `q` fails since there is no (introduction) rule for `q`.

For details about GCLA and its interpreter we refer to [Aronsson & Hallnäs] and

[Aronsson].
Planning becomes a special case of hypothetical reasoning in GCLA. We assume that there is a plan (e.g. a sequence of actions), and ask if it is possible to derive the desired state in the current program.

5. The GCLA Program

The actions in section 3 are almost directly coded into a GCLA program. The clause defining `if` is in fact not needed, but merely syntactic sugar.

```
------------
%% The functional if, coded in GCLA
if(X,Y) :- (X -> Y).

%% An action:
action(E,sit(S)) :- if(possible(E), perform(E,sit(S))).
action(E,action(X,S)) :-
     ((action(X,S) -> sit(Y)) -> action(E,sit(Y))).

%% Preconditions:
possible(stack(X,Y)) :-
     table(X), clear(X), clear(Y), (X = Y -> false).
possible(unstack(X,Y)) :-
     on(X,Y), clear(X).
possible(move(X,Y,Z)) :-
     on(X,Y), clear(X), clear(Z), (X = Z -> false).

perform(stack(X,Y),sit(S)) :-
     rem(table(X), rem(clear(Y),                    % delete list
          def(on(X,Y), sit(s(S))))).                % add list
perform(unstack(X,Y),sit(S)) :-
     rem(on(X,Y),                                   % delete list
          def(table(X),                             % add list
               def(clear(Y), sit(s(S))))).
perform(move(X,Y,Z),sit(S)) :-
     rem(on(X,Y), rem(clear(Z),                     % delete list
          def(clear(Y), true),                      % add list
               def(on(X,Z), sit(s(S))))).
```

```
%% Initial state:
on(a,b).
on(b,c).
clear(a).
table(c).
------------
```

The second clause of `action` is a substitution schema, saying that "if `sit(Y)` is derivable from `action(X,S)`, then `action(E,action(X,S))` is replaced by `action(E,sit(Y))`". This is a form of functional evaluation (see [Aronsson & Hallnäs]).
The def-constructions of the form `def(Rule, Goal)` should be read as "define `Rule` in the program for the rest of the execution, and execute `Goal`". `Rule` is removed after successful execution, or upon failure. `rem` does the opposite, that is, removes all rules that are equal to `Rule` (modulo variable names), so `rem` removes zero or more rules from the program. `def` and `rem` always succeed.
The initial state is the same as the one in fig. 1.

6. Planning

Generating a plan using the program in section 5 is accomplished by asking what should be assumed in order to derive the desired state. For example, to reach a state where the block `a` is on the table, the query `action(X,sit(0)) ⊢ table(a)` is asked to the program, and the system comes up with the solution `X = unstack(a,b)`. What happens is that we are looking for what assumption should be made in order to achieve the desired state `table(a)`.
Another way of looking on this query is to say that `action(X,sit(0))` is a function call, which should give as a result `table(a)`. Thus the call `action(X,sit(0)) ⊢ table(a)` is a form of equation solving; find an `X` such that the evaluation of `action(X,sit(0))` gives as value `table(a)` (although in this case we are using the program to pass the result `table(a)`).
Of course one can ask more complicated questions, like "find three actions that give as a result that block c stands on block b": `action(X,action(Y,action(Z,sit(0)))) ⊢ on(c,b)`. There are two solutions to this query, namely `Z = unstack(a,b), Y = unstack(b,c), X = stack(c,b)` and `Z = unstack(a,b), Y = move(b,c,a), X = stack(c,b)`. The query for generating a plan from the state in fig. 1 to the state in fig. 2 is (`action(X,action(Y,action(Z,sit(0)))) ⊢ on(c,b)`, `action(X,action(Y,action(Z,sit(0)))) ⊢ on(b,a)`) which gives as the only result `Z = unstack(a,b), Y = move(b,c,a), X = stack(c,b)`.
In the appendix there is a complete GCLA deduction of the query "find two actions that give as a result a state where `b` stands on the table": `action(X,action(Y,sit(0))) ⊢ table(b)`.

7. Simulation

Simulation is accomplished by instantiating the action part in the queries in section 6 and having uninstantiated states to the right of ⊢. This means that planning or simulating a sequence of actions becomes a question of instantiation in the query/goal.
A simple simulation query is "What stands on the table if the block a is unstacked"; `action(unstack(a,b),sit(0)) ⊢ table(X).` This is the "opposite" of the first question given in section 6. (We are using the same program as in the planning section, the program given in section 5.) The answer is of course `X = a` or `X = c`.
This can be seen as an evaluation of the function `action(unstack(a,b),sit(0))` in order to achieve a canonical value. When this canonical value is reached the simulation is finished and we can look at the state and see what has happened. By a small change in the GCLA rules it is possible to change the query to "During the simulation of `action(unstack(a,b),sit(0))`, what stands on the table?".
It is also possible to look at the query as "From the assumption `action(unstack(a,b),sit(0))`, does it follows that anything stands on the table?".
Another query is:

```
(action(stack(b,c),action(move(b,c,a),action(unstack(a,b),sit(0)))) ⊢
on(X,Y),
action(stack(b,c),action(move(b,c,a),action(unstack(a,b),sit(0)))) ⊢
on(Y,Z))
```

or in english " if block a is unstacked, block b is moved from block c to block a, and block c is stacked on block b, is there a pile of three blocks?". Depending on the order of the GCLA rules, the two possible answers, `X = a, Y = b, Z = c` and `X = c, Y = b, Z = a`, will come in a different order. The first answer is where no action has taken place, which will come first if the rules are ordered for "lazy evaluation", and the second one is where all the actions have taken place, which will come first if the actions are all evaluated before the on-terms are considered, "eager evaluation".

One can also simulate the first part, and then plan the rest of the execution to reach the desired state. For example, consider the query "If block a is first unstacked, and there can be at most one action more, is it possible to reach a state where block b stands on the table?"; `action(X,action(unstack(a,b),sit(0))) ⊢ table(b).` The correct answer is of course `X = unstack(b,c)` (see the appendix).
Another possibility is to plan the first part, and then simulate the rest of the query, which corresponds to knowing which action should take place in the end, but not knowing how to start, for example `action(unstack(b,c),action(X,sit(0))) ⊢ table(b)`, where the answer is `X = unstack(a,b)` (see the appendix).

8. Conclusion

In this report we have shown that the STRIPS planning system can be implemented in GCLA in a direct and natural way. Planning and simulation become one another's inverse, and is accomplished by the instantiation pattern in the queries. It is also

possible to mix simulation and planning.
This idea is now the basis for another project, where we are trying to build a system which generates plans for a construction site.

Acknowledgements
Much of the work presented here has been done in cooperation with Anette Gäredal, to whom I am very thankful. I also want to thank Lars Hallnäs, who came up with the initial idea, Rune Gustavsson for reading and correcting and suggesting improvements on this paper, and Vicki Carleson for improving my English.

References

[Aronsson & Hallnäs]: Martin Aronsson, Lars Hallnäs; *GCLA, Generalized Horn Clauses as a Programming Language*, SICS Research Report R88014

[Aronsson]: Martin Aronsson; *GCLA User's Manual*, SICS Technical Report T89012

[Genesereth & Nilsson]: Michael R. Genesereth, Nils J. Nilsson; *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann Publishers 1987

[Hallnäs & Schroeder-Heister]: Lars Hallnäs, Peter Schroeder-Heister; *A Proof-Theoretic Approach to Logic Programming, I. Generalized Horn Clauses*,SICS research Report R88005

[Nilsson]: Nils J. Nilsson; *Principles of Artificial Intelligence*, Springer-Verlag 1980

Appendix

```
def(clear(c), sit(s(0))) ⊢ table(b)
def(table(b), def(clear(c), sit(s(0)))) ⊢ table(b)
rem(on(b,c), def(table(b), def(clear(c), sit(s(0))))) ⊢ table(b)
perform(unstack(b,c),sit(s(0))) ⊢ table(b)
⊢ clear(b) , perform(unstack(b,c),sit(s(0))) ⊢ table(b)
⊢ on(X2,Y2) , ⊢ clear(X2) , perform(unstack(X2,Y2),sit(s(0))) ⊢ table(b)
possible(A) -> perform(A,sit(s(0))) ⊢ table(b)
if(possible(A),perform(A,sit(s(0)))) ⊢ table(b)
action(A,sit(s(0))) ⊢ table(b)
sit(s(0))) ⊢ sit(Z1) , action(A,sit(Z1))⊢table(b)
def(clear(b), sit(s(0))) ⊢ sit(Z1) , action(A,sit(Z1))⊢table(b)
def(table(a), def(clear(b), sit(s(0)))) ⊢ sit(Z1) , action(A,sit(Z1))⊢table(b)
rem(on(a,b), def(table(a), def(clear(b), sit(s(0))))) ⊢ sit(Z1) , action(A,sit(Z1))⊢table(b)
perform(unstack(a,b),sit(S)) ⊢ sit(Z1) , action(A,sit(Z1))⊢table(b)
⊢ clear(a) , perform(unstack(a,b),sit(S)) ⊢ sit(Z1) , action(A,sit(Z1))⊢table(b)
⊢ on(X,Y) , ⊢ clear(X) , perform(unstack(X,Y),sit(S)) ⊢ sit(Z1) , action(A,sit(Z1))⊢table(b)
⊢ possible(E) perform(E,sit(S)) ⊢ sit(Z1) , action(A,sit(Z1))⊢table(b)
possible(E)->perform(E,sit(S)) ⊢ sit(Z1) , action(A,sit(Z1))⊢table(b)
if(possible(E),perform(E,sit(S)))⊢sit(Z1) , action(A,sit(Z1))⊢table(b)
action(Y,sit(0)) ⊢ sit(Z1) , action(A,sit(Z1)) ⊢ table(b)
⊢ (action(Y,sit(0)) -> sit(Z1)) , action(A,sit(Z1)) ⊢ table(b)
((action(Y,sit(0)) -> sit(Z1)) -> action(A,sit(Z1))) ⊢ table(b)
action(A,action(Y,sit(0))) ⊢ table(b)
```